



US009459992B2

(12) **United States Patent**
Spycher et al.

(10) **Patent No.:** **US 9,459,992 B2**
(45) **Date of Patent:** **Oct. 4, 2016**

(54) **SYSTEM AND METHOD OF DEBUGGING
MULTI-THREADED PROCESSES**

(56) **References Cited**

U.S. PATENT DOCUMENTS

(71) Applicant: **Synopsys, Inc.**, Mountain View, CA
(US)

5,819,093 A 10/1998 Davidson et al.
5,953,530 A 9/1999 Rishi et al.
6,042,614 A 3/2000 Davidson et al.

(72) Inventors: **Matthias Spycher**, Santa Clara, CA
(US); **Dietmar Petras**, Langerwehe
(DE)

(Continued)

(73) Assignee: **Synopsys, Inc.**, Mountain View, CA
(US)

OTHER PUBLICATIONS

(*) Notice: Subject to any disclaimer, the term of this
patent is extended or adjusted under 35
U.S.C. 154(b) by 145 days.

Milos Prvulovic et al.; ReEnact Using Thread-Level Speculation
Mechanisms to Debug Data Races in Multithreaded Codes; 2003
IEEE; 12 pages; <[http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=
&arnumber=1206993](http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1206993)>.*

(Continued)

(21) Appl. No.: **14/165,541**

Primary Examiner — Thuy Dao

(22) Filed: **Jan. 27, 2014**

Assistant Examiner — Cuong V Luu

(74) *Attorney, Agent, or Firm* — Fenwick & West LLP

(65) **Prior Publication Data**

US 2014/0196014 A1 Jul. 10, 2014

(57) **ABSTRACT**

A system and method of debugging a multi-threaded process with at least one running thread and at least one suspended thread is disclosed. Embodiments utilize a blocking function to block the thread of a process while other threads are allowed to run. The blocking function may be executed in a suspended thread by a debugger under control of a thread blocking controller. The other threads may implement inter-process communication channels for enabling communication between the process and another application. A simulated user interface (UI) of a debugger enables interaction with users while a hardware simulation thread is blocked, where blocking of the hardware simulation thread may be implemented by a thread blocking component implemented externally to the debugger. Where a thread blocking controller is implemented within the debugger, a debugger UI may interact with a user while the hardware simulation thread is blocked and interprocess communication threads are running.

Related U.S. Application Data

(62) Division of application No. 12/001,238, filed on Dec.
10, 2007, now Pat. No. 8,683,444.

(60) Provisional application No. 60/874,436, filed on Dec.
11, 2006.

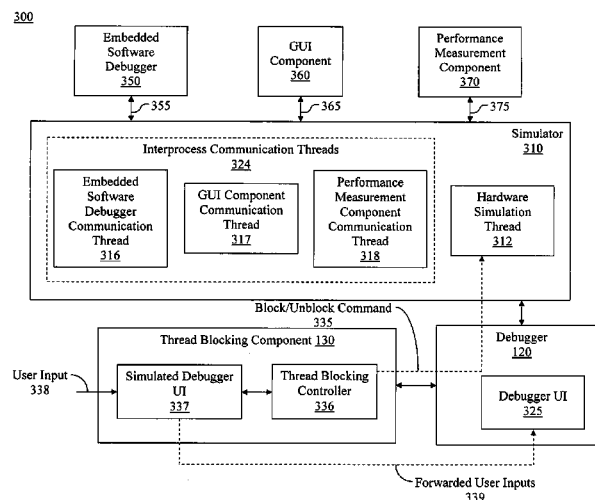
(51) **Int. Cl.**
G06F 9/44 (2006.01)
G06F 11/36 (2006.01)

(52) **U.S. Cl.**
CPC **G06F 11/3664** (2013.01); **G06F 11/3636**
(2013.01)

(58) **Field of Classification Search**
CPC G06F 11/3664; G06F 11/3636; G06F
9/455; G06F 17/5009; G06F 11/362; G06F
9/526

See application file for complete search history.

19 Claims, 6 Drawing Sheets



(56)

References Cited**U.S. PATENT DOCUMENTS**

6,378,124	B1	4/2002	Bates et al.	
6,587,967	B1 *	7/2003	Bates	G06F 11/3664 714/35
6,681,384	B1	1/2004	Bates et al.	
6,957,436	B1 *	10/2005	Mihic	718/107
6,978,399	B2	12/2005	Bates et al.	
7,171,653	B2	1/2007	Albrecht	
7,426,735	B2	9/2008	Bliss et al.	
7,441,233	B1 *	10/2008	Orndorff	G06F 9/4443 717/107
7,613,599	B2 *	11/2009	Bade	G06F 17/5022 703/14
7,716,031	B2 *	5/2010	Vanspauwen et al.	703/13
7,904,886	B2	3/2011	Dufour et al.	
2002/0059054	A1 *	5/2002	Bade	G06F 17/5022 703/20
2003/0192032	A1	10/2003	Andrade et al.	
2004/0054944	A1	3/2004	Bates et al.	
2004/0194067	A1	9/2004	Lien et al.	
2004/0250244	A1 *	12/2004	Albrecht	717/135
2005/0034102	A1 *	2/2005	Peck	G06F 11/3648 717/124
2005/0044552	A1	2/2005	Bliss et al.	
2005/0071814	A1	3/2005	Aguilar et al.	
2006/0195821	A1 *	8/2006	Vanspauwen	G06F 11/3664 717/124
2006/0218535	A1	9/2006	Delmonte et al.	
2006/0253842	A1 *	11/2006	Pees	G06F 11/3632 717/129
2007/0011656	A1 *	1/2007	Kumamoto	G06F 11/3664 717/124
2007/0043698	A1	2/2007	Short et al.	
2007/0094642	A1 *	4/2007	Dove	G06F 8/34 717/124
2007/0168984	A1	7/2007	Heishi et al.	
2007/0180430	A1	8/2007	Farchi et al.	
2007/0234295	A1 *	10/2007	Dufour	G06F 11/3644 717/124
2008/0263521	A1	10/2008	Neumann et al.	
2008/0301644	A1 *	12/2008	Drepper	G06F 9/44589 717/124
2010/0058296	A1	3/2010	Nattinger	

OTHER PUBLICATIONS

Patric Bohrer et al.; Mambo a full system simulator for the PowerPC architecture; 2004 ACM; pp. 8-12; <<http://dl.acm.org/citation.cfm?id=1054910>>.*

Anthony-Tmg Nguyen et al.; The augmint multiprocessor simulation toolkit for Intel x86 architecture; 1996 IEEE; pp. 486-490; <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=563597>>.*

Isabel A. Nepomuceno-Chamorro; A Java Simulator for Basic Transition P Systems; 2004 Citeseer; pp. 309-315; <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.225.4187&rep=rep1&type=pdf>>.*

M. Adda et al.; A Simulator for a Multithreaded Processor; 1999 Citeseer; 4 pages; <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.3202&rep=rep1&type=pdf>>.*

Haixia Wang et al.; Acceleration Techniques for Chip-Multiprocessor Simulator Debug; 2006 Springer; pp. 509-515; <http://link.springer.com/chapter/10.1007/11859802_51>.*

Ghosh, A., A Hardware-Software Co-simulator for Embedded System Design and Debugging, Mitsubishi Electr, Sep. 1995, pp. 155-164, <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=486217&tag=1>.

Klein, Russell, Miami: A Hardware Software Co-Simulation Environment, 1996 IEEE, pp. 173-177, <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=506802>.*

Becker, David, An Engineering Environment for Hardware/software Co-Simulation, 1992 IEEE, pp. 129-134, <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=227848>.*

Wang, Perry H., Helper Threads via Virtual Multithreading, 2004 IEEE, pp. 74-82, <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=138816>>.*

Prvulovic, Milos, ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes, 2003 IEEE, 12 pages, <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1206993>>.*

Talavera, G., Hardware-Software Debugging Techniques for Reconfigurable Systems-on-Chip, 2004 IEEE, pp. 1402-1407, <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1490767>>.*

Ghosh, Sumit et al., An Asynchronous Distributed Approach for the Simulation of Behavior-Level Models on Parallel Processors, 1992 IEEE, pp. 639-652, <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=388044>>.*

Bohrer, Patrick, Mambo A full system simulator for the PowerPC Architecture, 2004 ACM, pp. 8-12, <<http://dl.acm.org/citation.cfm?id=1054910>>.*

Nguyen, Anthony-Tmg et al., The augmint multiprocessor simulation toolkit for Intel x86 architectures, 1996 IEEE, pp. 486-490, <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=563597>>.*

Yoo, Sungjoo et al., Optimistic Distributed Timed Cosimulation Based on Thread Simulation Model, 1998 IEEE, pp. 71-75, <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=666240>>.*

Raghavan, Ram et al., Multiprocessor System Verification Through Behavioral Modeling and Simulation, 1995 IEEE, pp. 395-402, <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=472462>>.*

Zilic, Zeljko et al, Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug, 2006 IEEE, 6 pages, <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4380831>>.*

Parson, D. et al., "A Framework for Simulating Heterogeneous Virtual Processors," Proceedings of the 32nd Annual Simulation Symposium, 1999, pp. 58-67. May be retrieved at URL: <http://www.ieee.org/portal/innovate/search/article_details.html?article=7664155>.*

United States Ex Parte Quayle Action, U.S. Appl. No. 12/001,238, May 3, 2008, 6 pages.

United States Office Action, U.S. Appl. No. 12/001,238, Oct. 9, 2012, 19 pages.

United States Office Action, U.S. Appl. No. 12/001,238, Jun. 11, 2012, 18 pages.

United States Office Action, U.S. Appl. No. 12/001,238, Jan. 9, 2012, 19 pages.

United States Office Action, U.S. Appl. No. 12/001,238, Aug. 12, 2011, 18 pages.

* cited by examiner

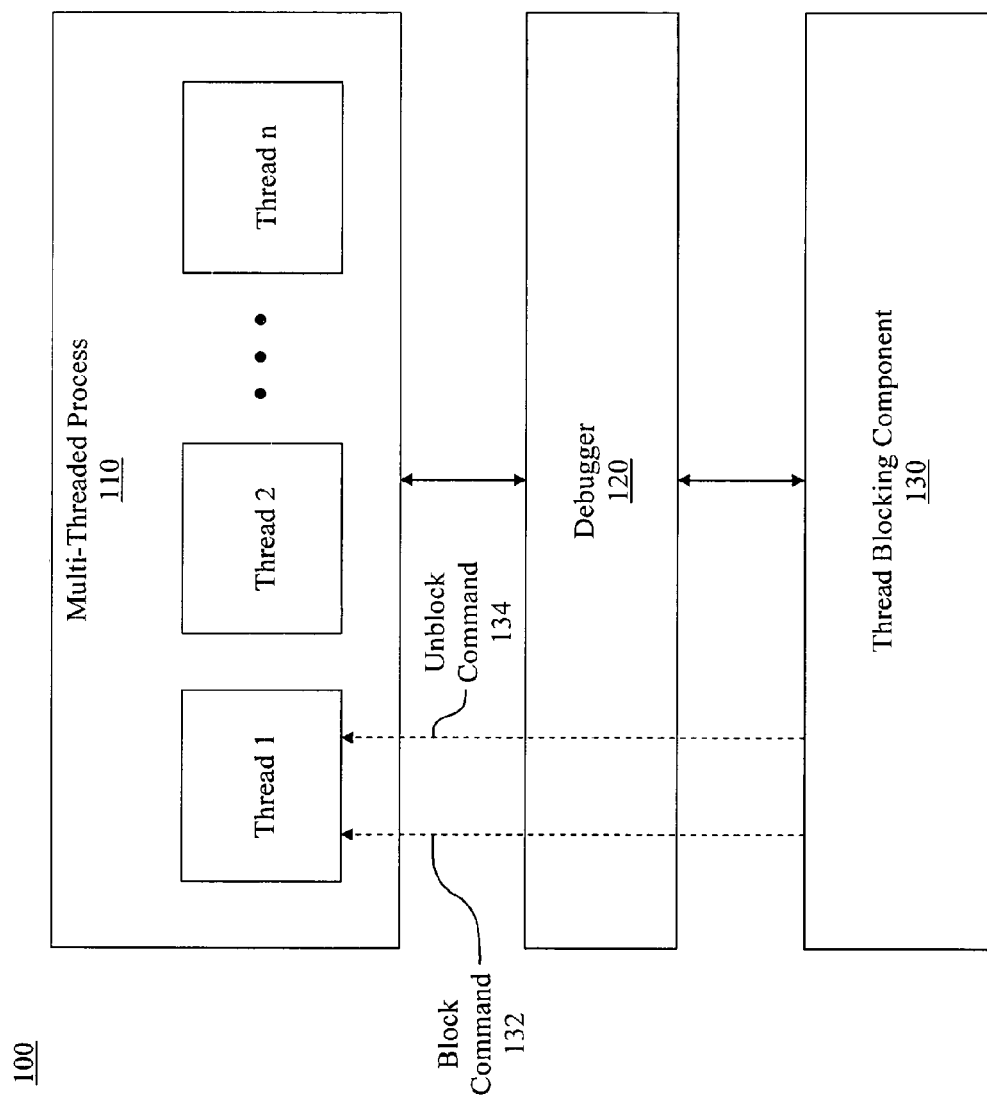
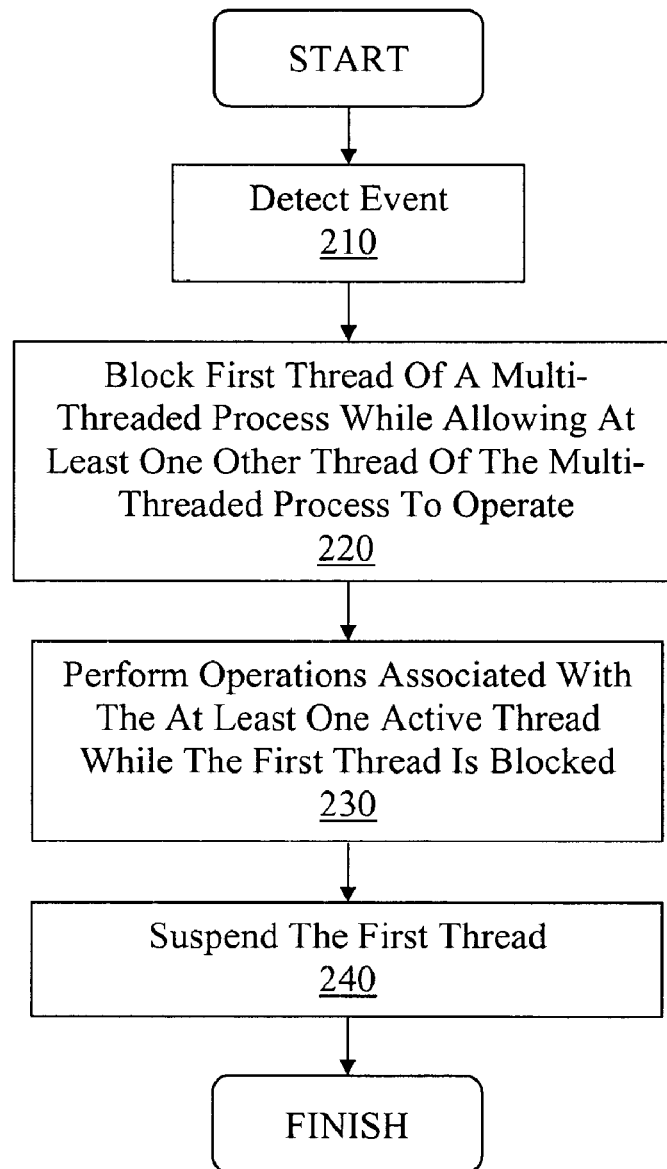
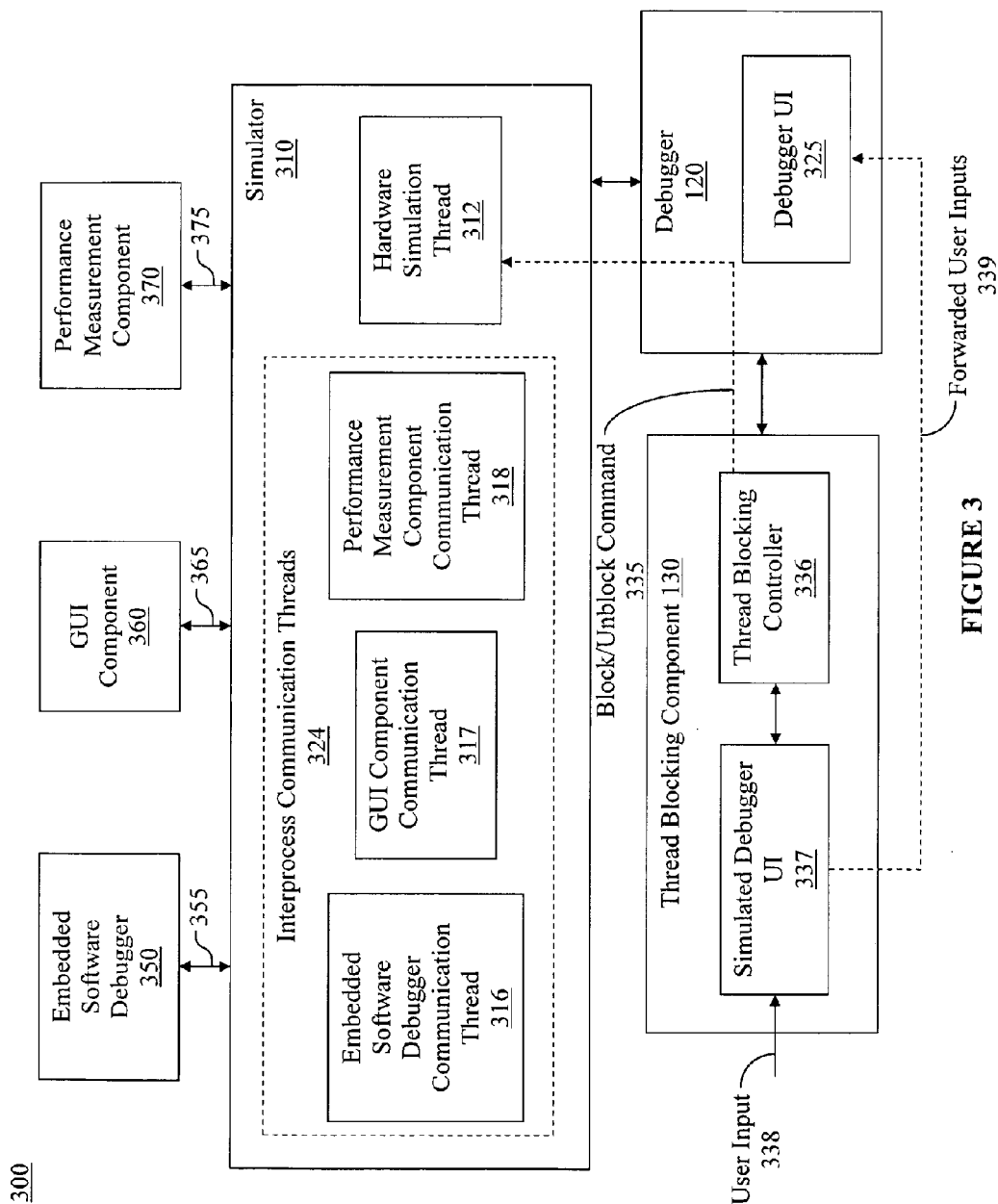


FIGURE 1

200**FIGURE 2**



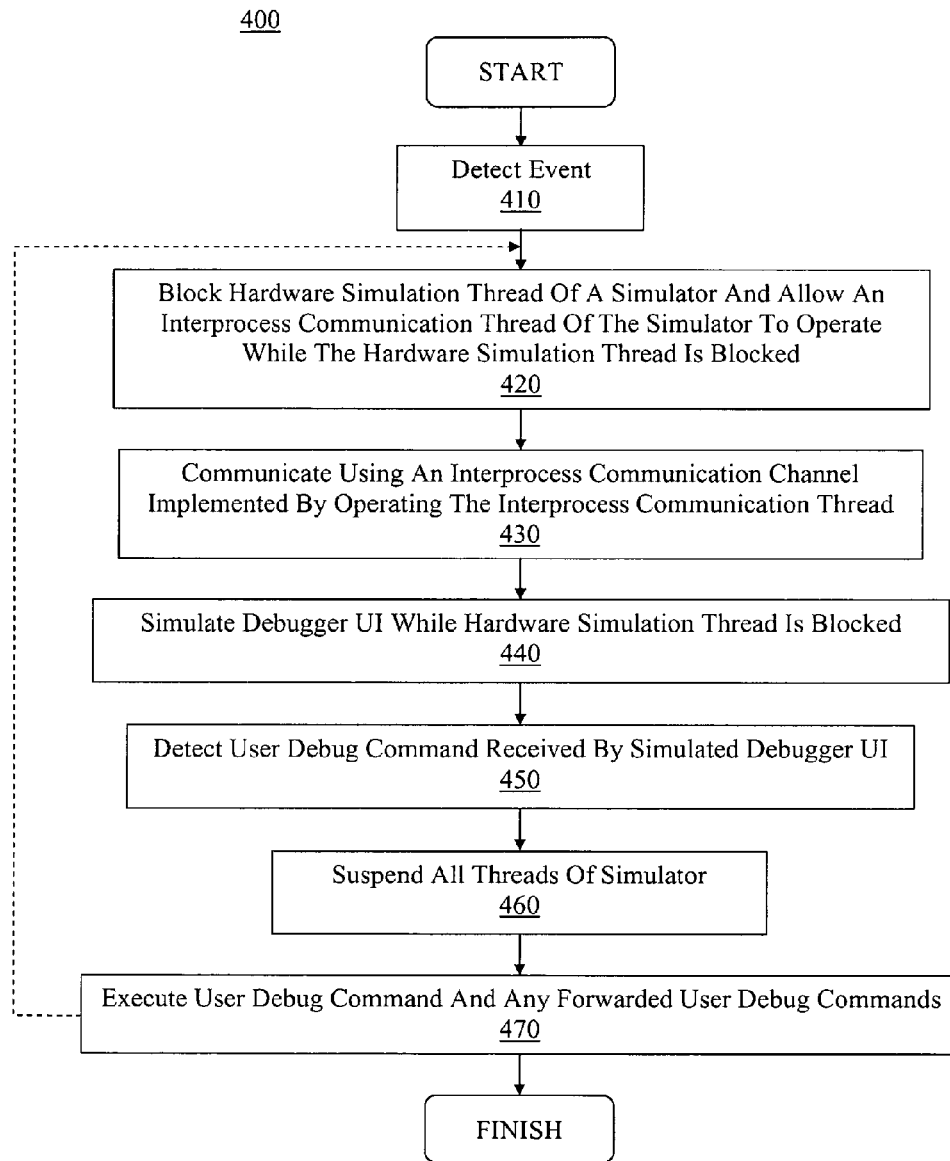


FIGURE 4

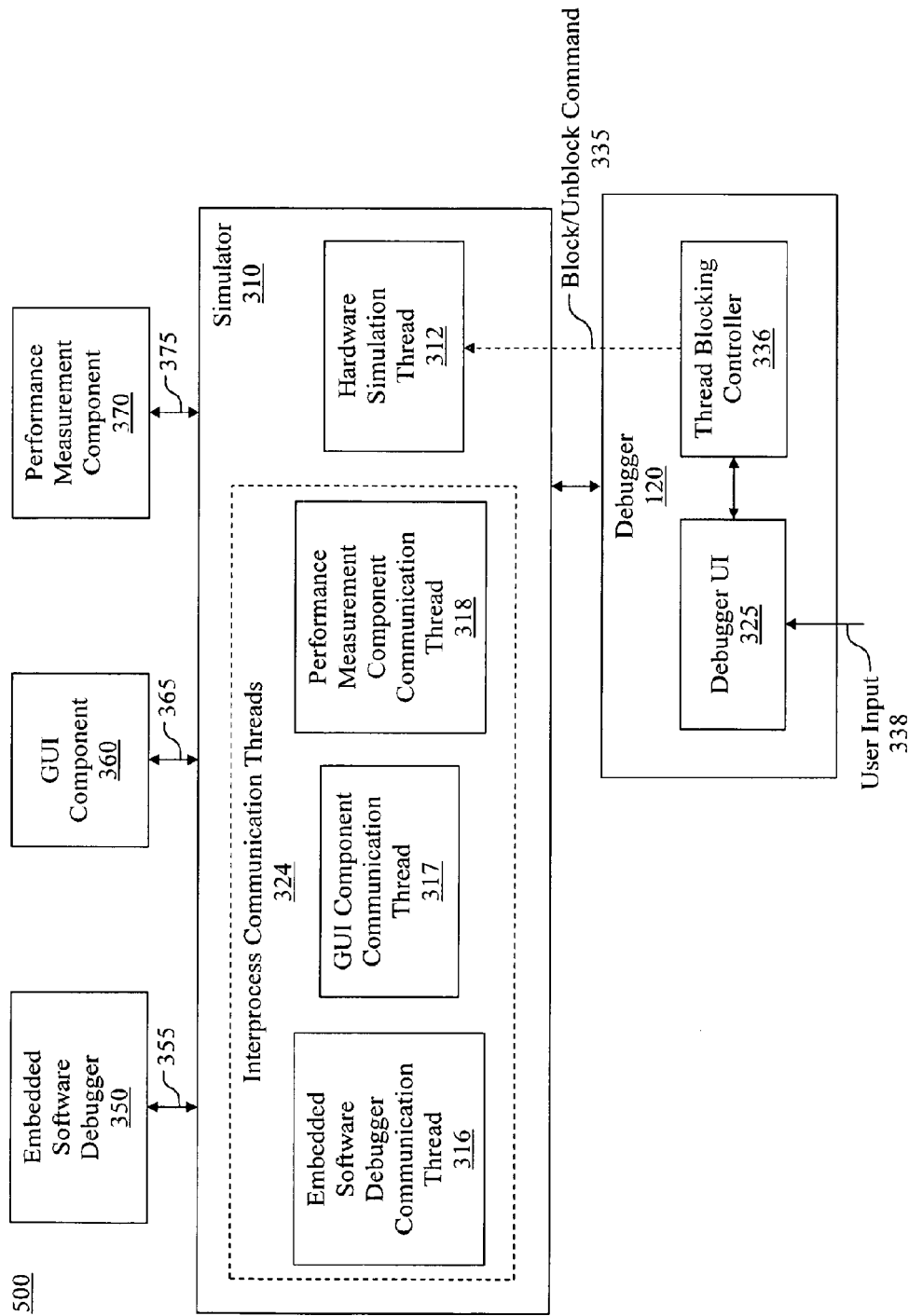


FIGURE 5

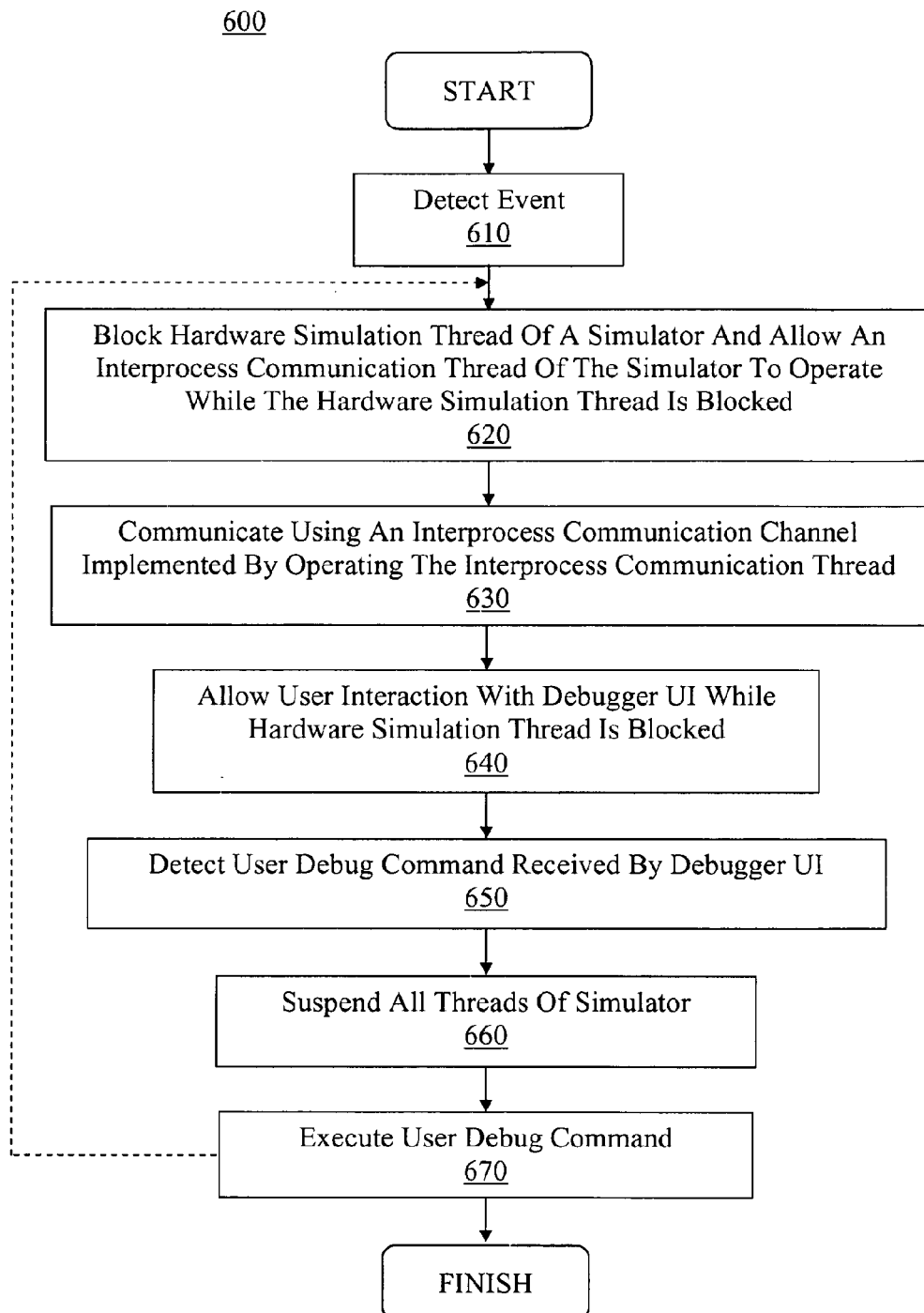


FIGURE 6

SYSTEM AND METHOD OF DEBUGGING MULTI-THREADED PROCESSES

CROSS-REFERENCE TO RELATED APPLICATIONS

The present application is a divisional of U.S. patent application Ser. No. 12/001,238 filed on Dec. 10, 2007, entitled "SYSTEM AND METHOD OF DEBUGGING MULTI-THREADED PROCESSES," which claims the benefit of U.S. Provisional Patent Application No. 60/874,436, filed Dec. 11, 2006, entitled "SYSTEM AND METHOD FOR ENHANCED DEBUGGING OF VIRTUAL PLATFORMS." Both applications are incorporated herein by reference in their entirety and for all purposes.

BACKGROUND OF THE INVENTION

Debugging of multi-threaded hardware simulation processes is generally carried out by suspending all threads of the hardware simulation process before beginning debugging operations. Most debuggers rely upon the operating system to assist in debugging the hardware simulation process, and most operating systems (e.g., Windows, Linux, etc.) require the suspension of all threads. Once all the threads of the multi-threaded hardware simulation process are suspended, a user may step through or otherwise examine simulation results from the simulation of the hardware model to locate and fix bugs in the hardware model.

Since conventional debugging systems debug multi-threaded hardware simulation processes when all threads are suspended, they provide users limited debugging functionality for multi-threaded hardware simulation processes which rely upon the suspended threads to communicate with other applications (i.e., using interprocess communication). For example, where the state of the simulated platform model depends upon information in another application, the conventional debugger may return an incomplete or incorrect picture of the simulated hardware state for debugging purposes.

Additionally, the ability of conventional debugging systems to debug embedded software (e.g., applications run on the simulated hardware component) is also limited when all threads of the hardware simulator are suspended. Debugging of embedded software generally requires an embedded software debugger to access the state of the underlying simulated platform. Accordingly, by suspending interprocess communication threads of the hardware simulation, the interprocess communication channels used by the embedded software debugger to access the state of the simulated platform are severed. As such, conventional debugging systems provide embedded software debuggers with limited access to the state of the simulated platform, thereby resulting in incomplete and potentially inaccurate information for debugging of the embedded software.

SUMMARY OF THE INVENTION

Accordingly, a need exists for a debugging system which enables the running of one or more threads of a process while another thread of the process is blocked. More specifically, a need exists for a debugging system which blocks a hardware simulation thread while allowing interprocess communication threads to run. Further, a need exists for a debugging system which provides a simulation process access to applications coupled via interprocess communication channels. A need also exists for a debugging system

which provides embedded software debuggers access to the simulated platform state to enable debugging of embedded software. Embodiments of the present invention provide novel solutions to these needs and others as described below.

Embodiments are directed to a system and method of debugging a multi-threaded process with at least one running thread and at least one suspended thread. More specifically, embodiments utilize a function call on the thread to be suspended for implementing a blocking function (e.g., executing a loop with a guard variable and a sleep function, blocking on a thread control element such as a semaphore, condition variable, mutex, etc.) to block the thread (e.g., a hardware simulation thread) of a process (e.g., a simulator for simulating a hardware component or platform) while other threads are allowed to run. The blocking function may be executed in a suspended thread by a debugger under control of a thread blocking controller in one embodiment. Where the other threads comprise interprocess communication threads (e.g., to enable communication with other processes or applications such as embedded software debugger applications, graphical user interfaces or visualization applications, performance measurement applications, etc.), embodiments enable communication over interprocess communication channels implemented by the interprocess communication threads. Accordingly, embodiments enable debuggers (e.g., hardware debuggers, embedded software debuggers, etc.) and other components (e.g., applications coupled to the simulator via interprocess communication channels, etc.) to access the simulated platform state or otherwise communicate via the interprocess communication channels, thereby improving the debugging of software (e.g., embedded software, etc) and simulated hardware components (e.g., a simulated hardware platform, etc.).

Embodiments also provide a simulated user interface (UI) of a debugger for enabling interaction with users while a hardware simulation thread is blocked, where the blocking of the hardware simulation thread may be implemented by a thread blocking component implemented externally to the debugger. Once a user debug command is detected, all threads of the simulator may be suspended (e.g., using an unblock function which aborts the blocking of the blocking function such that the blocking function returns) to enable execution of the user debug command accessed from the simulated debugger UI. In one embodiment, user inputs to the simulated debugger UI may be forwarded to the debugger while all threads of the simulator are suspended. The hardware simulation thread may thereafter be blocked (e.g., using a blocking function) while at least one interprocess communication thread is allowed to operate. In one embodiment, the blocking of the hardware simulation thread may occur automatically after a predetermined period of time without detection of a user debug command (e.g., input via the simulated debugger UI).

In one embodiment where a thread blocking controller is implemented within the debugger, a UI of the debugger may interact with a user while the hardware simulation thread is blocked and interprocess communication threads are running. Once a user debug command is detected, all threads of the simulator may be suspended (e.g., using an unblock function which aborts the blocking of the blocking function such that the blocking function returns) to enable execution of the user debug command accessed from the simulated debugger UI. The debugger UI may continue to process user inputs while all threads of the simulator are suspended. The hardware simulation thread may thereafter be blocked while at least one interprocess communication thread is allowed to operate (e.g., to enable communication with other processes

3

or applications such as embedded software debugger applications, graphical user interfaces or visualization applications, performance measurement applications, etc.). In one embodiment, the blocking of the hardware simulation thread may occur automatically after a predetermined period of time without detection of a user debug command (e.g., input to the debugger UI).

In one embodiment, a method of debugging a system includes, responsive to a breakpoint event, using a debugger tool to block a first thread of a multi-threaded hardware simulator of a hardware platform, wherein the multi-threaded hardware simulator includes a plurality of threads including the first thread for simulating the hardware platform and a plurality of second threads for implementing interprocess communications between the hardware simulator and other applications within the system. While the first thread is blocked, a user interface mechanism is implemented, external to the debugger tool, for receiving a user debug command. The method also includes, responsive to the user debug command, performing the following: suspending all threads of the multi-threaded hardware simulator; executing the user debug command on the debugger tool; after execution of the user debug command, blocking again the first thread; and repeating the implementing the user interface mechanism for receiving a user debug command. The other applications may be selected from a group consisting of: a visualization application; a performance measurement application; and an embedded software debugger application. An inferior call on the first thread may be issued causing a blocking function to be executed to block the first thread. The blocking function may be selected from a group consisting of: executing a loop with a guard variable; and blocking on a thread control element. The thread control element may be selected from a group consisting of: a semaphore; a wait condition; a conditional variable; and a mutex.

In another embodiment, a method of debugging a system includes, in response to a detected event, blocking a first thread of a simulator using a debugger communicatively coupled to the simulator, the simulator further including a second thread for implementing an interprocess communication channel between the simulator and an additional process. The second thread is allowed to operate while the first thread is blocked. The method also includes communicating using the interprocess communication channel while the first thread is blocked. The first thread is suspended. The blocking the first thread may include executing a blocking function in the first thread, and wherein the suspending the first thread may include executing an unblock function to abort the blocking function. The blocking function may be selected from a group consisting of a sleep function, a semaphore, a wait condition, a conditional variable, and a mutex.

In yet another embodiment, a system includes a multi-threaded simulator including a first thread and a second thread, the second thread for implementing an interprocess communication channel between the simulator and an additional process. A debugger is communicatively coupled to the multi-threaded simulator, the debugger for blocking the first thread in response to a detected event, the debugger further for allowing the second thread to operate while the first thread is blocked, the debugger further for suspending all threads of said multi-threaded simulator. A controller is communicatively coupled to the debugger and for initiating the blocking and the suspending performed by the debugger. The additional process may be a process of the system. The simulator may include a hardware simulator, and wherein

4

the first thread includes a hardware simulation thread. The debugger may be operable to execute a blocking function in the first thread to implement the blocking, and wherein the debugger may further be operable to implement the suspending by executing an unblock function to abort the blocking function. The blocking function may be selected from a group consisting of a sleep function, a semaphore, a wait condition, a conditional variable, and a mutex.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements.

FIG. 1 shows an exemplary debugging system for debugging a multi-threaded process with at least one active thread and a suspended thread in accordance with one embodiment of the present invention.

FIG. 2 shows an exemplary process for debugging a system in accordance with one embodiment of the present invention.

FIG. 3 shows an exemplary debugging system for implementing interprocess communication while a hardware simulation thread is blocked in accordance with one embodiment of the present invention.

FIG. 4 shows an exemplary process for debugging a simulated component in accordance with one embodiment of the present invention.

FIG. 5 shows an exemplary debugging system for implementing interprocess communication while a hardware simulation thread is blocked using a debugger with an internal thread blocking controller in accordance with one embodiment of the present invention.

FIG. 6 shows an exemplary process for debugging a simulated component using a debugger with an internal thread blocking controller in accordance with one embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

Reference will now be made in detail to embodiments of the present invention, examples of which are illustrated in the accompanying drawings. While the present invention will be discussed in conjunction with the following embodiments, it will be understood that they are not intended to limit the present invention to these embodiments alone. On the contrary, the present invention is intended to cover alternatives, modifications, and equivalents which may be included with the spirit and scope of the present invention as defined by the appended claims. Furthermore, in the following detailed description of the present invention, numerous specific details are set forth in order to provide a thorough understanding of the present invention. However, embodiments of the present invention may be practiced without these specific details. In other instances, well-known methods, procedures, components, and circuits have not been described in detail so as not to unnecessarily obscure aspects of the present invention.

Notation and Nomenclature

Some regions of the detailed descriptions which follow are presented in terms of procedures, logic blocks, processing and other symbolic representations of operations on data bits within a computer memory. These descriptions and

representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. In the present application, a procedure, logic block, process, or the like, is conceived to be a self-consistent sequence of steps or instructions leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, although not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated in a computer system.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing the terms such as “aborting,” “accepting,” “accessing,” “adding,” “adjusting,” “analyzing,” “applying,” “assembling,” “assigning,” “balancing,” “blocking,” “calculating,” “capturing,” “combining,” “comparing,” “collecting,” “creating,” “debugging,” “defining,” “depicting,” “detecting,” “determining,” “displaying,” “establishing,” “executing,” “flipping,” “generating,” “grouping,” “hiding,” “identifying,” “initiating,” “interacting,” “modifying,” “monitoring,” “moving,” “outputting,” “performing,” “placing,” “presenting,” “processing,” “programming,” “querying,” “removing,” “repeating,” “resuming,” “sampling,” “simulating,” “sorting,” “storing,” “subtracting,” “suspending,” “tracking,” “transforming,” “unblocking,” “using,” or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system’s registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

Embodiments of the Invention

FIG. 1 shows exemplary debugging system 100 for debugging a multi-threaded process with at least one active thread and a blocked thread in accordance with one embodiment of the present invention. As shown in FIG. 1, multi-threaded process 110 comprises multiple threads (e.g., Thread 1 through Thread n), where Thread 1 is blocked by debugger 120 while Thread 2 through Thread n are allowed to run. As such, debugger 120 may debug multi-threaded process 110 while Thread 1 is blocked and Thread 2 through Thread n are running, thereby extending the capabilities of debugger 120 (e.g., by enabling debugger 120 to access additional information using one or more active threads, to perform additional functions implemented by the active threads, etc.). Additionally, the presence of active threads in multi-threaded process 110 may enable other systems or devices (e.g., an embedded software debugger operable to debug embedded software running on top of multi-threaded process 110, etc.) to access the state of multi-threaded process 110 while thread 1 is blocked.

Multi-threaded process 110 may comprise any piece of software code comprising instructions which can be run on a computer system. Multi-threaded process 110 may comprise an application or program for performing one or more functions. In one embodiment, multi-threaded process 110 may implement a hardware simulator capable of simulating

a hardware and/or software platform. Additionally, in one embodiment, multi-threaded process 100 may conform to the SystemC standard.

In one embodiment, Thread 1 may comprise an operating system (OS) thread which is provided by or with the support of the OS (e.g., the kernel of the OS, other portions of the OS, etc.). OS threads may be distinguished from light-weight threads or quick threads which are implemented without support of the OS. Additionally, in one embodiment, Thread 1 may comprise an OS thread which comprises multiple sub-threads, where one or more of the sub-threads are implemented by at least one light-weight thread. Further, where multi-threaded process 110 implements a SystemC simulator for performing SystemC simulations, multi-threaded process 110 may comprise multiple SystemC threads (e.g., which may be implemented by way of light-weight threads) which may be executed within an OS thread (e.g., Thread 1) in one embodiment.

Debugger 120 may comprise any debugger capable of accessing multi-threaded process 110 (e.g., results from execution of multi-threaded process, etc.). Debugger 120 may enable the location and/or correction of bugs within multi-threaded process 110. Additionally, debugger 120 may comprise a hardware debugger and/or system-level debugger in one embodiment.

Blocking of Thread 1 may be controlled by thread blocking component 130 using block command 132. Block command 132 may effectively block Thread 1, thereby reducing functionality implemented by Thread 1 and/or reducing access to Thread 1 (e.g., by other components of system 100, other systems, etc.). Block command 132 may be implemented by controlling debugger 120 (e.g., using thread blocking component 130) to call a blocking function (e.g., by issuing an “inferior call”) for execution in Thread 1 in one embodiment. The blocking function may comprise a loop with a guard variable, a thread control element operable to block a thread (e.g., a semaphore, wait condition, condition variable, mutex, etc.), etc. In one embodiment, calling the blocking function in the thread to be suspended may simultaneously block Thread 1 while activating or otherwise making operational one or more of Thread 2 through Thread n. The blocking of Thread 1 and/or activation of at least one of Thread 2 through Thread n may be performed after a suspension of all threads of multi-threaded process 110 (e.g., in response to encountering a breakpoint or other event associated with multi-threaded process 110, debugger 120, etc.) in one embodiment.

Thread blocking component 130 may also suspend threads of multi-threaded process 110 (e.g., Thread 1, Thread 2, Thread n, etc.) using unblock command 134. Unblock command 134 may abort the blocking of the blocking function (e.g., implemented using blocking command 132) such that the blocking function returns. In this manner, all threads of multi-threaded process 110 may be suspended, in one embodiment, such that debug commands may be executed by debugger 120. Unblock command 134 may be implemented by controlling debugger 120 (e.g., using thread blocking component 130) to call an unblock function for execution in a thread of system 100 (e.g., Thread 1, Thread 2, Thread n, etc.) in one embodiment. Additionally, a TCP socket may be used as an IPC channel for aborting the blocking function in one embodiment.

In one embodiment, block command 132 and unblock command 134 may be implemented by performing a loop with a guard variable “Blocker” and sleep function in accordance with the following exemplary code:

```

volatile bool Blocker;
void blockFunction( ) {
    const int duration_ms = 50;
    Blocker = true;
    while (Blocker == true) {
        Sleep(duration_ms);
    }
}
void breakFunction( ) {
    Blocker = false;
}

```

As such, the function “blockFunction()” implements block command **132** and the function “breakFunction()” implements unblock command **134**.

In another embodiment, block command **132** and unblock command **134** may be performed by a mutex and a condition variable (or wait condition) in accordance with the following exemplary code:

```

volatile bool Blocker;
ConditionVariable BlockCondition;
Mutex BlockMutex;
void blockFunction( ) {
    BlockMutex.lock( );
    Blocker = true;
    while (Blocker == true) {
        BlockCondition.wait(BlockMutex);
    }
    BlockMutex.unlock( );
}
void breakFunction( ) {
    BlockMutex.lock( );
    Blocker = false;
    BlockCondition.signal( );
    BlockMutex.unlock( );
}

```

It should be appreciated that more than one thread of multi-threaded process **110** may be blocked in one embodiment. Additionally, it should be appreciated that multi-threaded process **110** may comprise any number of active or running threads which operate while at least one thread of multi-threaded process **110** is blocked. Further, it should be appreciated that thread blocking component **130** (or one or more components of thread blocking component **130**) may be alternatively implemented (e.g., within debugger **120**, within a system external to system **100**, etc.) in other embodiments.

FIG. **2** shows exemplary process **200** for debugging a system in accordance with one embodiment of the present invention. As shown in FIG. **2**, step **210** involves detecting an event. The detected event may comprise a breakpoint associated with a multi-threaded process (e.g., **110**). The detected event may cause control to be transferred to a debugger (e.g., **120**), which is communicatively coupled to the multi-threaded process (e.g., **110**), such that the debugger may examine the state of the multi-threaded process and/or perform debugging operations on the multi-threaded process in one embodiment. Additionally, in one embodiment, all threads of the multi-threaded process (e.g., **110**) may be suspended in response to the event detected in step **210**.

Step **220** involves blocking a first thread (e.g., Thread **1** depicted in FIG. **1**) of a multi-threaded process (e.g., **110**) while allowing at least one other thread (e.g., Thread **2**, Thread **n**, etc. depicted in FIG. **1**) of the multi-threaded process to operate. The thread may be blocked using a block command (e.g., **132**) as discussed above with respect to FIG. **1** in one embodiment.

As shown in FIG. **2**, step **230** involves performing operations associated with at least one active thread while the first thread is blocked. For example, where the at least one active thread is operable to implement an interprocess communication (IPC) channel (e.g., to enable communication between the multi-threaded process and another system, to enable communication between the multi-threaded process and an application within system **100**, etc.), communications may be transmitted and received in accordance with instructions executed by the at least one active thread. In one embodiment, the IPC channels may enable communication with or otherwise provide access to embedded software, embedded software debuggers, etc. Alternatively, other operations may be performed by execution of operations within the at least one active thread while the first thread is blocked.

Step **240** involves suspending the first thread. The first thread may be suspended using an unblock command (e.g., **134**) as discussed above with respect to FIG. **1** in one embodiment. Step **240** may be performed in response to a completion of operations performed in step **230** in one embodiment. Alternatively, step **240** may be performed in response to detection of a debug command (e.g., a user debug command, machine-generated debug command, etc.), thereby enabling a debugger (e.g., **120**) to execute the debug command while the first thread is suspended. Accordingly, in one embodiment, step **240** may also involve load balancing between the time in which a debugger (e.g., **120**) is blocking the first thread and the time in which the debugger is executing a command (e.g., a user debug command, machine-generated debug command, etc.). In this fashion, neither of the two periods is allowed to get crowded out. And in another embodiment, step **240** may be performed automatically after a predetermined time period without detection of user interaction or input of additional debug commands. And in another embodiment, all threads of the multi-threaded process (e.g., **110**) may be suspended in step **240**.

FIG. **3** shows exemplary debugging system **300** for implementing interprocess communication while a hardware simulation thread is blocked in accordance with one embodiment of the present invention. As shown in FIG. **3**, simulator **310** may represent one embodiment of multi-threaded process **110** of FIG. **1**. Simulator **310** comprises hardware simulation thread **312** for implementing a simulation of a component model (e.g., a processor, microprocessor, etc.), where the component model may also be referred to as a platform (e.g., hardware platform, hardware/software platform, etc.), virtual platform, etc. Simulator **310** also comprises IPC threads **324** such as embedded software debugger communication thread **316** (e.g., for implementing IPC channel **355** between embedded software debugger **350** and simulator **310**), graphical user interface (GUI) component communication thread **317** (e.g., for implementing IPC channel **365** between GUI component **360** and simulator **310**), and performance measurement component communication thread **318** (e.g., for implementing IPC channel **375** between performance measurement component **370** and simulator **310**). As such, the components or applications (e.g., **350-370**) and simulator **310** may communicate using their respective IPC channels (e.g., **355-375**) implemented by respective active IPC threads (e.g., **316-318**) while hardware simulation thread **312** is suspended. Additionally, IPC channels **355-375** may provide debugger or other components of system **300** access to applications coupled to

simulator 310 (e.g., embedded software debugger 350, GUI component 360, performance measurement component 370, etc.).

In one embodiment, simulator 310 may comprises a SystemC simulator for performing SystemC simulations. Hardware simulation thread 312 may comprise an OS thread which comprises multiple SystemC threads. One or more of the SystemC threads may be implemented using at least one light-weight thread (or quick thread) in one embodiment.

As shown in FIG. 3, thread blocking component 130 comprises thread blocking controller 336 for controlling the blocking and suspending of hardware simulation thread 312. Thread blocking controller 336 may utilize debugger 120 to initiate execution of block/unblock commands 335 (e.g., block command 132 and unblock command 134 as discussed above with respect to FIG. 1 above) in hardware simulation thread 312, thereby enabling thread blocking controller 336 to control blocking and suspending of hardware simulation thread 336. Additionally, when blocking hardware simulation thread 312, a TCP socket may be used as an IPC channel for aborting a blocking function used to implement block command (e.g., 132) in one embodiment.

Thread blocking component 130 may also comprise simulated debugger user interface (UI) 337 for simulating debugger UI 325 of debugger 120. Debugger UI 325 and/or simulated debugger UI 337 may comprise a GUI, command-line user interface, or the like. While hardware simulation thread 312 is blocked, simulated debugger UI 337 may interact with users and accept user input 338. In one embodiment, debugger UI 325 may be unable to interact with a user and/or accept user inputs while hardware simulation thread is blocked (e.g., since the backend of debugger 120 may only support one operation at a time and is currently busy implementing the blocking function), and thus, simulated debugger UI 337 may enable system 300 to interact with users and/or accept user inputs in this situation. Additionally, simulated debugger UI 337 may function similarly to debugger UI 325 (e.g., such that a user may be unaware that he or she is interacting with a simulated UI as opposed to debugger UI 325) in one embodiment.

Where user input 338 comprises a user debug command for execution on debugger 120, simulated debugger UI 337 may notify thread blocking controller 336 that a user debug command has been received. Thread blocking controller may then issue an unblock command (e.g., 134, 335, etc.) to suspend one or more threads of simulator 310 in order to process the user debug command. In one embodiment, the suspension of the one or more threads of simulator 310 may be delayed (e.g., if a predetermined time period has not elapsed since the last suspension of hardware simulation thread 312). Once the one or more threads are suspended, debugger 120 may execute the user debug command, which may comprise accessing the state (e.g., stored in registers associated with simulator 310 and/or hardware simulation thread 312, etc.) of simulator 310, locating bugs associated with the simulated component, fixing located bugs, etc. While the one or more threads are suspended, user inputs 338 may be forwarded (e.g., represented by forwarded user inputs 339) from simulated debugger UI 337 to debugger UI 325 for processing by debugger 120. While the one or more threads are suspended, interprocess communication via IPC channels 355-375 may be suspended in one embodiment. Thereafter, hardware simulation thread 312 may be blocked (e.g., by debugger 120 under control of thread blocking controller 336), where, in one embodiment, blocking of hardware simulation thread 312 may resume or enable IPC communication via IPC channels 355-375.

Thread blocking controller 336, in conjunction with simulated debugger UI 337 and debugger 120, may automatically block and/or suspend threads of simulator 310 based upon one or more conditions or events in one embodiment. For example, debugger 120 may notify simulated debugger UI 337 when IPC communications or other operations associated with active threads (e.g., IPC threads 324) are completed, and therefore enable thread blocking controller 336 to automatically suspend all threads of simulator 310 (e.g., to prepare for execution of debug operations by debugger 120). Alternatively, threads of simulator 310 may be automatically suspended in response to detection of a debug command (e.g., a user debug command, machine-generated debug command, etc.) by simulated debugger UI 337 and/or thread blocking component 130, thereby enabling debugger 120 to execute the debug command while threads of simulator 310 are suspended. And in another embodiment, thread blocking controller 336 and/or simulated debugger UI 337 may monitor the time since the last user input and/or user debug command (e.g., input to simulated debugger UI 337), thereby enabling thread blocking controller 336 to automatically suspend threads of simulator 310 after a predetermined time period without detection of user interaction or input of additional debug commands.

In one embodiment, simulated debugger UI 337 may receive machine-input debug commands (not depicted in FIG. 3) in addition to or in place of user debug commands. The machine-input debug commands may be accessed from another system or device (e.g., external to system 300) and/or from a component of system 300. It should be appreciated that system 300 may process machine-input debug commands (e.g., by appropriately blocking hardware simulation thread 312 and appropriately suspending threads of simulator 310 to enable debugger to execute the machine-input debug commands) which are fed to system 300 at a relatively high rate (e.g., 1000 inputs per second or higher) in one embodiment. Accordingly, components of system 300 (e.g., thread blocking controller 336, simulated debugger UI 337, debugger 120, etc.) may implement load balancing between the time in which debugger 120 is blocking hardware simulation thread 312 and the time in which debugger 120 is executing a command (e.g., a user debug command, machine-generated debug command, etc.) so that neither of these two periods is allowed to get crowded out.

As shown in FIG. 3, embedded software debugger 350 may comprise a source-level debugger for debugging embedded software (e.g., operable to run on the hardware component being simulated by simulator 310), where the embedded software may comprise GUI component 360 and performance measurement component 370 in one embodiment. GUI component 360 may comprise a visualization application for implementing a GUI on a device or system using the simulated hardware component. Performance measurement component 370 may comprise an application for measuring the performance of one or more components of the device or system using the simulated hardware component. Accordingly, implementing IPC channels 355-375 (e.g., through activation of IPC threads 324) may enable embedded software debugger 350 to access and examine the state (e.g., stored within registers associated with simulator 310 and/or hardware simulation thread 312) of the underlying simulation performed by simulator 310 (e.g., by executing hardware simulation thread 312), and therefore, implement debugging of embedded software (e.g., GUI component 360, performance measurement 370, etc.) communicatively coupled to simulator 310.

11

Additionally, thread blocking component 130 may be implemented using multiple threads (e.g., one or more threads to implement thread blocking controller 336, and one or more other threads to implement simulated debugger UI 337). Alternatively, thread blocking controller 336 and simulated debugger UI 337 may be implemented by the same thread which implements a state machine. Accordingly, operation of each component (e.g., thread blocking controller 336 and simulated debugger UI 337) may be controlled by the state machine in one embodiment.

Although FIG. 3 shows only three applications (e.g., embedded software debugger 350, GUI component 360, performance measurement component 370, etc.) coupled to simulator 310 via IPC channels (e.g., 355-375), it should be appreciated that a larger or smaller number of applications may be coupled to simulator 310 in other embodiments. Additionally, although FIG. 3 depicts only four threads within simulator 310, it should be appreciated that simulator 310 may comprise a larger or smaller number of threads in other embodiments. It should also be appreciated that simulator 310 may comprise a larger or smaller number of blocked, suspended and/or active threads in other embodiments. For example, thread blocking controller 336 may control blocking, suspension and/or activation of any number of threads within simulator 310 in one embodiment. Further, it should be appreciated that thread blocking component 130 may comprise a different number of components in other embodiments.

FIG. 4 shows exemplary process 400 for debugging a simulated component in accordance with one embodiment of the present invention. As shown in FIG. 4, step 410 involves detecting an event. The detected event may comprise a breakpoint associated with a multi-threaded simulator (e.g., 310). The detected event may cause control to be transferred to a debugger (e.g., 120), which is communicatively coupled to the multi-threaded simulator (e.g., 310), such that the debugger may examine the state of the simulation and/or perform debugging operations on the multi-threaded simulator in one embodiment. Additionally, in one embodiment, all threads of the simulator (e.g., 310) may be suspended in response to the event detected in step 410.

Step 420 involves blocking a hardware simulation thread (e.g., 312) of the simulator (e.g., 310) and allowing an IPC thread (e.g., 316, 317, 318, etc.) to operate while the hardware simulation thread is blocked. The thread may be blocked using a block command (e.g., 132) as discussed above with respect to FIG. 1 in one embodiment. The block command may be initiated by a thread blocking controller (e.g., 336) of a thread blocking component (e.g., 130), where the thread blocking controller is operable to control a coupled debugger (e.g., 120) to execute a blocking function in the hardware simulation thread (e.g., 312) in one embodiment.

As shown in FIG. 4, step 430 involves communicating using an IPC channel (e.g., 355, 365, 375, etc.) implemented by the operating IPC thread (e.g., 316, 317, 318, etc.). For example, a debugger (e.g., 120) for debugging a simulated hardware component running on the simulator (e.g., 310) may access an application (e.g., embedded software debugger 350, GUI component 360, performance measurement component 370, etc.) coupled to the simulator (e.g., 310). Alternatively, an application (e.g., embedded software debugger 350, etc.) coupled to the simulator may access a state of the simulation (e.g., performed by simulator 310) and/or one or more other applications (e.g., GUI component 360, performance measurement component 370, etc.) communicatively coupled to the simulator (e.g., 310).

12

Step 440 involves simulating a debugger UI while the hardware simulation thread is blocked. The simulated debugger UI (e.g., 337) may simulate a debugger UI (e.g., 325) of the debugger (e.g., 120) executing the blocking of the hardware simulation thread (e.g., 312). The simulated debugger UI (e.g., 337) may be implemented externally to the simulator (e.g., 310). Additionally, the simulated debugger UI (e.g., 337) may be operable to receive inputs from a user (e.g., user interactions with the simulated UI, user debug commands, etc.) and/or from a system/device (e.g., a machine-input debugger command, etc.).

As shown in FIG. 4, step 450 involves detecting a user debug command received by the simulated debugger UI (e.g., 337). The user debug command may be detected by the simulated debugger UI (e.g., 337) itself and/or a thread blocking controller (e.g., 336) coupled to the simulated debugger UI (e.g., 337).

Step 460 involves suspending all threads of the simulator (e.g., 310). The threads of the simulator (e.g., 310) may be suspended using an unblock command (e.g., 134) as discussed above with respect to FIG. 1 in one embodiment. Step 460 may be performed in response to a completion of IPC communications performed in step 430 in one embodiment. Alternatively, step 460 may be performed in response to detection of a debug command (e.g., a user debug command, machine-generated debug command, etc.) performed in step 450, thereby enabling a debugger (e.g., 120) to execute the debug command while threads of the simulator (e.g., 310) are suspended. Accordingly, in one embodiment, step 460 may also involve load balancing between the time in which a debugger (e.g., 120) is blocking the hardware simulation thread (e.g., 312) and the time in which the debugger is executing a command (e.g., a user debug command, machine-generated debug command, etc.). And in another embodiment, step 460 may be performed automatically after a predetermined time period without detection of user interaction or input of additional debug commands.

Suspension of threads of the simulator (e.g., 310) in step 460 may also comprise aborting of the blocking function (e.g., implemented to block the hardware simulation thread in step 420). In one embodiment, aborting the blocking function may be performed by changing a value of a guard variable of a loop implementing the blocking function, signaling a semaphore implementing the blocking function, signaling a wait condition implementing the blocking function, signaling a condition variable implementing the blocking function, unlocking a mutex implementing the blocking function, or some combination thereof. Additionally, a TCP socket may be used as an IPC channel for aborting the blocking function in one embodiment. Further, additional user debug commands received while the threads of the simulator (e.g., 310) are suspended may be forwarded to the debugger (e.g., 120) by the simulated debugger UI (e.g., 337).

Step 470 involves executing the user debug command and any forwarded user debug commands. The debug commands may be executed by the debugger (e.g., 120). Thereafter, steps 420 through 470 may be repeated (e.g., as indicated by the dotted arrow). Alternatively, process 400 may conclude after step 470.

It should be appreciated that the execution of debug commands in step 470 may be interleaved with one or more steps of process 400. For example, if a debug command received in step 450 may be executed in step 470 before an additional debug command is forwarded. As such, the forwarded debug commands may be executed upon a future

13

suspension of threads of the simulator (e.g., 310). Additionally, it should be appreciated that one or more steps of process 400 may be controlled or implemented by a state machine. The state machine may be implemented by a single thread (e.g., as discussed above with respect to FIG. 3) in one embodiment.

FIG. 5 shows exemplary debugging system 500 for implementing interprocess communication while a hardware simulation thread is blocked using a debugger with an internal thread blocking controller in accordance with one embodiment of the present invention. As shown in FIG. 5, system 500 may operate similarly to system 300 of FIG. 3, except that thread blocking controller 336 is implemented within debugger 120 in system 500 as depicted in FIG. 5. As such, debugger UI 325 may interact with a user and accept user inputs 338 directly. Additionally, debugger UI 325 may interact with a user and accept user inputs 338 while hardware simulation thread 312 is blocked. Debugger UI 325 may also accept machine-input debugger commands from a device or system external to system 500, a component of system 500, etc.

FIG. 6 shows exemplary process 600 for debugging a simulated component using a debugger with an internal thread blocking controller in accordance with one embodiment of the present invention. As shown in FIG. 6, steps 610-630 may be performed analogously to steps 410-430 of FIG. 4 in one embodiment.

Step 640 involves allowing user interaction with a debugger UI (e.g., 325) while the hardware simulation thread (e.g., 312) is blocked. For example, user inputs (e.g., 338) may be accepted and the debugger UI may be updated in response to the user inputs in one embodiment.

As shown in FIG. 6, step 650 involves detecting a user debug command received by the debugger UI (e.g., 325). The user debug command may be detected by the debugger UI (e.g., 325) itself and/or a thread blocking controller (e.g., 336) coupled to the debugger UI (e.g., 325).

Step 660 involves suspending all threads of the simulator (e.g., 310) and may be performed analogously to step 460 of FIG. 4. Step 670 involves executing the user debug command. The debug commands may be executed by the debugger (e.g., 120). Thereafter, steps 620 through 670 may be repeated (e.g., as indicated by the dotted arrow). Alternatively, process 600 may conclude after step 670.

In the foregoing specification, embodiments of the invention have been described with reference to numerous specific details that may vary from implementation to implementation. Thus, the sole and exclusive indicator of what is, and is intended by the applicant to be, the invention is the set of claims that issue from this application, in the specific form in which such claims issue, including any subsequent correction. Hence, no limitation, element, property, feature, advantage, or attribute that is not expressly recited in a claim should limit the scope of such claim in any way. Accordingly, the specification and drawings are to be regarded in an illustrative rather than a restrictive sense.

What is claimed is:

1. A method of debugging a platform, said method comprising:

detecting a breakpoint event of a simulator that is communicatively coupled to a debugger, the simulator comprising a first thread simulating the platform, a second thread implementing a first interprocess communication channel between the simulator and an embedded software debugger that is different than the debugger and a third thread implementing a second interprocess communication channel between the simu-

14

lator and a component that is different than the debugger and different than the embedded software debugger and external to the simulator;

blocking, using the debugger and responsive to detecting the breakpoint event, the first thread simulating the platform while allowing the second thread implementing the first interprocess communication channel to operate and allowing the third thread implementing the second interprocess communication channel to operate; and

communicating using the second thread implementing the first interprocess communication channel and the third thread implementing the second interprocess communication channel while the first thread simulating the platform is blocked, the embedded software debugger accessing a state of the platform using the second thread during the communicating.

2. The method of claim 1, further comprising:

responsive to input of a debug command to the debugger, suspending the first thread, the second thread and the third thread;

executing the debug command using the debugger while the first thread, the second thread and the third thread are suspended; and

blocking again, after execution of the debug command, the first thread simulating the platform while allowing the second thread implementing the first interprocess communication channel to operate and allowing the third thread implementing the second interprocess communication channel to operate.

3. A method of debugging a platform, said method comprising:

detecting an event of a simulator that is communicatively coupled to a debugger, the simulator comprising a first thread simulating the platform, a second thread implementing a first interprocess communication channel between the simulator and a first component that is different than the debugger and external to the simulator and a third thread implementing a second interprocess communication channel between the simulator and a second component different than the debugger and different than the first component and external to the simulator;

blocking, using the debugger and responsive to detecting the event, the first thread simulating the platform while allowing the second thread implementing the first interprocess communication channel to operate and allowing the third thread implementing the second interprocess communication channel to operate; and

communicating using the second thread implementing the interprocess communication channel and the third thread implementing the second interprocess communication channel while the first thread simulating the platform is blocked.

4. The method of claim 3, further comprising:

suspending the first thread, the second thread and the third thread after communicating using the second thread and the third thread;

executing a debug command using the debugger while the first thread, the second thread and the third thread are suspended; and

blocking again, after execution of the debug command, the first thread simulating the platform while allowing the second thread implementing the first interprocess communication channel to operate and allowing the third thread implementing the second interprocess communication channel to operate.

15

5. The method of claim 4, wherein the first thread, the second thread and the third thread are suspended responsive to input of the debug command to the debugger.

6. The method of claim 4, wherein the first thread, the second thread and the third thread are suspended responsive to completion of communications using the second thread implementing the first interprocess communication channel and the third thread implementing the second interprocess communication channel.

7. The method of claim 4, wherein the first thread, the second thread and the third thread are suspended responsive to a predetermined time period without input of a user debug command.

8. The method of claim 3, wherein the event is a breakpoint event, and blocking the first thread comprises:

blocking, using the debugger and responsive to detecting the breakpoint event, the first thread simulating the platform while allowing the second thread implementing the first interprocess communication channel to operate and allowing the third thread implementing the second interprocess communication channel to operate.

9. The method of claim 3, wherein communicating using the second thread comprises:

accessing, by the first component, a state of the platform using the second thread implementing the first interprocess communication channel.

10. The method of claim 3, wherein the platform simulated by the first thread is a component model.

11. The method of claim 3, wherein blocking the first thread comprises:

executing a blocking function in the first thread to block the first thread.

12. The method of claim 11, wherein said blocking function is selected from a group consisting of a sleep function, a semaphore, a wait condition, a conditional variable, and a mutex.

13. The method of claim 3, wherein the second thread implements an interprocess communication channel between the simulator and an embedded software debugger.

14. The method of claim 3, wherein the third thread implements an interprocess communication channel between the simulator and a graphical user interface application.

15. The method of claim 3, wherein the third thread implements an interprocess communication channel between the simulator and a performance measurement application.

16

16. A computer memory storing instructions for debugging a platform, the instructions when executed by a computer system cause the computer system to:

detect an event of a simulator that is communicatively coupled to a debugger, the simulator comprising a first thread simulating the platform and a second thread implementing a first interprocess communication channel between the simulator and a first component that is different than the debugger and external to the simulator and a third thread implementing a second interprocess communication channel between the simulator and a second component that is different than the debugger and different than the first component and external to the simulator;

block, using the debugger and responsive to detecting the event, the first thread simulating the platform while allowing the second thread implementing the first interprocess communication channel to operate and allowing the third thread implementing the second interprocess communication channel to operate; and

communicate using the second thread implementing the first interprocess communication channel and the third thread implementing the second interprocess communication channel while the first thread simulating the platform is blocked.

17. The computer memory of claim 16, the instructions further causing the computer system to:

suspend the first thread, the second thread and the third thread after communicating using the second thread and the third thread;

execute a debug command using the debugger while the first thread, the second thread and the third thread are suspended; and

block again, after execution of the debug command, the first thread simulating the platform while allowing the second thread implementing the first interprocess communication channel to operate and allowing the third thread implementing the second interprocess channel to operate.

18. The computer memory of claim 17, wherein the first, the second thread and the third thread are suspended responsive to input of the debug command to the debugger.

19. The computer memory of claim 16, wherein the second thread implements an interprocess communication channel between the simulator and an embedded software debugger.

* * * * *